REMARKS

Claims 1-14 are pending in the application. Claims 1 and 4 are being amended.

All outstanding requirements will now be addressed in the order they appear in the Office Action mailed March 28, 2008.

Specification

2. The disclosure and claims 1-7 are objected to because of the phraseology "declared as ROM memory". Moreover, the Office Action alleges that the disclosure lacks support in order to justify the use of such language or to convey some insight that would help alleviate what appears to be unreasonable deviation from well-accepted meaning of the concepts implicated. As stated in the Office Action, the specification mentions "that it is necessary to inform linker" that addresses where the uncompressed data resides are permanent ROM-type, not amounting to any utility performing any declaration.

Applicants respectfully disagree and submit that declaring memory segments types for linking software is common knowledge in the art. Where a linker is to be configured with parameters, for example in assembler or c programming, a common method is to use a linker parameters file *.prm, which contains memory declarations in the SEGMENTS section. This is a typical approach known to a programmer skilled in the art.

For example, on the second page of the enclosed presentation (Appendix A) available at http://www.coe.montana.edu/ee/lameres/courses/ee371_fall06/lecture_notes/ee371_lecture_15.pdf from a college-lever course on microprocessor systems, one finds that:

,,... In the Linker Parameter File (*.prm), we setup memory and place these SECTIONS into the proper spot
Setting up Memory (in the Linker Parameter File)
- we use the keyword SECTIONS or SEGMENTS to setup memory map
- within this setup, we have 4 types of memory that we can setup

1) READ_ONLY
2) READ_WRITE (...) ..."

In case of c programming language the same configuring scheme applies. It is obvious to treat a declaration of memory sections before linking process as preparing a linker parameters file. There is no utility to make such declarations. This may be done in any text editor available.

Exemplary declaration in c programming language can be found, for example, at http://www.coe.montana.edu/ee/courses/EE/EE371/pdffiles/linker_2.pdf – page 7 from a college-level course (Appendix B):

Example 5-3 Linker Parameter File
NAMES
END
SECTIONS
RAM = READ_WRITE 0x0800 TO 0x0FFF;
/* unbanked FLASH ROM */
ROM_C000 = READ_ONLY 0xC000 TO 0xFEFF;
END

Additional information on this topic can be commonly found at http://uhaweb.hartford.edu/jmhill/suppnotes/CodeW12/param.htm (Appendix C)

Therefore, Applicants respectfully submit that memory segments declaration before a linking process is common general knowledge and the extrinsic evidence provided confirms such approach. Any skilled in the art programmer faced with a linker setup, requiring memory segments declaration, would make use of a *.prm file scheme.

Claim Objections

3. Claim 6 stands objected to because of an extraneous 'a' between 'function common to' and 'the decompression'. Applicants have deleted the extraneous 'a' to obviate the Examiner's objection.

Claim Rejections – 35 USC § 101

4. Claim 7 stands rejected under 35 U.S.C. 101 because the claimed invention is directed to a non-statutory subject matter.

Applicants respectfully disagree and submit that a method for updating software in a data signal receiver having a processor and data exchange interfaces, RAM, ROM, NV-RAM and a non-volatile memories linked to the processor is not directed to a non-statutory subject matter. The useful, concrete, and tangible result of the inventive method is the lower requirement of memory or bandwidth capacity in case of storage or transmission of software. Specifically, the instant specification teaches in a relevant part that "...These are the FLASH 150, and RAM 160 memories. These memories store the programs controlling the digital television decoder functions, including the compressed loader, which stores a method for controlling the non-volatile memory, for instance the Flash type memory 150, or the decoder's RAM type memory 160..."

Claim Rejections – 35 USC § 102

5-6. Claim 1, 5-10, 13-14 stand rejected under 35 U.S.C. 102(b) as being anticipated by Euroloader, "Technical Specification of a European Loader for Multimedia Terminals for Cable and Cable Modems", December 2001.

Applicants have amended claim 1 by modifying the previously presented feature that the data signal receiver comprises means for declaring a section of the RAM memory as ROM type memory, prior to a software linking process. This feature is not known in the prior art.

Claim Rejections – 35 USC § 103

7-10. Claim 2 stands rejected under 35 U.S.C. 103(a) as being unpatentable over Euroloader, "Technical Specification of a European Loader for Multimedia Terminals for Cable and Cable Modems", December 2001 in view of Defosse et al, US Publ. No. 2003/0097474 whereas claims 3-4, 11-12 stand rejected under 35 U.S.C. 103(a) as being unpatentable over Euroloader, "Technical Specification of a European Loader for Multimedia Terminals for Cable and Cable Modems", December 2001 in view of Tuttle US. Publ. No. 5,325,377.

The Office Action alleges that 'decompressing program of the loader' is present in the specification of Euroloader with reference to the MD5 calculation and that the decompression of the loader is covered by the successful verification of the loader according to the Euroloader specification.

The Office Action further alleges that "...checking whether a generated checksum matches a given MD5 checksum reads on regenerating a crc - recalculating a checksum inherently entails crc for data in a decompressed form - from the decompressed data recently received to check its integrity...".

Applicants still respectfully disagree as the specification of Euroloader explicitly uses MD5 hash and checksum, which does not inherently entail any compression. Hashing and compression are two separate processes, which may exist together but are not essential for each other as the examiner implies without any grounds.

Requirement of checking integrity of files does not arise from using compression. As the cited diagram 7 shows, the purpose is security. The integrity check may arise from uncertain transmission paths.

Furthermore MD5 is a cryptographic hash function with a 128-bit hash value. It is employed in a wide variety of security applications, and is also commonly used to check the integrity of files. An MD5 hash is typically expressed as a 32-character hexadecimal number. In particular, it is used in Euroloader to perform verification of the loader. The purpose of MD5 use in Euroloader is thus to check the integrity of the loader.

In contrast, the present invention relates to decompressing program of the loader, where checking of the integrity may be only an optional and non-essential part of the decompression algorithm. The purpose of the present invention is thus compressing (limiting the size of) the loader, which is not present in the Euroloader.

Therefore, Euroloader does not read on the decompression present in claim 1 and requires different techniques to operate.

Moreover the quoted successful verification of the Euroloader has nothing to do with the decompression of claim 1 per se. It is, however, true that a decompression process may be subsequently verified in order to establish whether it has been successful, which in fact was described in the original specification with reference to Fig. 3 step 308. It is nevertheless evident that the verification step is optional and was not included in the claim 1 as filed.

Additionally, the specification of the Euroloader does not mention that a section of RAM has to be declared as ROM as opposed to the present invention. The purpose of this declaration is described in paragraph 24 of the application: "...Uncompressed loaders may be located in any available memory segment. In the case of the compressed loader, permanent memory addresses are used, in order to assure proper references to locations containing variables, constants or pointers to functions..."

Because the specification of the Euroloader does not mention, by any means, a possibility of adding decompression to the loader and as has been presented, by the above

explanations, the prior art does not suggest the desirability of the claimed invention, the Applicants respectfully submit that the subject matter of the independent claim 1 is patentable.

Therefore, Applicants respectfully request withdrawal of the rejection with respect to claims 1 – 14 as amended in view of the explanation presented above.

CONCLUSION

In view of the foregoing amendments and remarks, Applicants submit that the pending claims are in condition for allowance. Early and favorable reconsideration is respectfully solicited. No authorization to charge deposit account is given and any prior outstanding authorizations are hereby rescinded. Any fees dues will be remitted via EFS-Web until further notice.

Respectfully Submitted,

Customer Number: **33,794**

/Matthias Scholl/

Dr. Matthias Scholl, Esq.
Reg. No. 54,947

Date: June 20, 2008

EE 371 – Microprocessor Hardware & Software Systems

Lecture #15

- **Agenda**
  1. Assembler Expressions
- **Announcements**
  1. Read Chapter 6

Assembler Expressions

Expressions

- Until now, everything in class has been referencing absolute assembler directives (i.e., ORG).
- In Lab, we are using Relocatable Assembler directives.
- In Relocatable Assemblers, multiple source codes are assembled and the linker combines the various object codes into one *absolute* file which contains the memory map information.
- the LINKER parameter file (*.prm) is the file that contains the information on how to map the code into memory.

Linker Setup

Groupings

We can group our code into 4 main sub sections :

ROM
1) Code Section
2) Constant Section

RAM
3) Variable Data Section
4) STACK Section

SECTION

In our Source Code (*.asm), we use the keyword SECTION to signify blocks of code that can be re-mapped by the linker.

ex) MyCode: SECTION

MyConst: SECTION

MyData: SECTION

Linker setup

SECTION cont...

In the Linker Parameter File (*.prm), we setup memory and place these SECTIONS into the proper spot

Setting up Memory (in the Linker Parameter File)

- we use the keyword SECTIONS or SEGMENTS to setup memory map

- within this setup, we have 4 types of memory that we can setup

1) READ_ONLY
2) READ_WRITE
3) NO_INIT (not used for now)
4) PAGED (not used for now)

- the syntax for defining these blocks of memory is:

<name> = (type) (start_addr) TO (end_addr)

- at the end of this setup, we use the keyword END to notify the linker we are done defining.

Linker Setup

SECTION cont…

Example)

```
SECTION
        RAM       = READ_WRITE   0X0800  to  0X0FFF
        ROM_4000  = READ_ONLY    0X4000  to  0X7FFF
        RAM_C000  = READ_ONLY    0XC000  to  0XFEFF
END
```

Linker Setup

SECTION cont…

Placement in Memory (in the Linker Parameter File)

- we use the keyword PLACEMENT to locate our sections in the memory map
- the syntax for defining these blocks of memory is:

  <section_name> INTO (memory_name)

- we need to include two default types for the linker (DEFAULT_ROM, DEFAULT_RAM) in case there are sections of code that don't have a location specified.
- we use the keyword END to notify the linker we are done placing
- the order of the section lists dictates the order they will be put in memory
- the keyword SSTACK represents the STACK section, we must define its placement also.

Linker Setup

Setting up the STACK

We define the size of the stack using the keyword STACKSIZE (size)

ex) STACKSIZE 0x0100

Including this in the Linker Parameter File does two things:

1) sets the size of the stack (in this case to 256 bytes)
2) creates a symbol for the end of the stack (+1) to initialize the SP to called

__SEG_END_STACK (NOTE : there are TWO underscores)

Setting up the Reset VECTOR

A vector is an address that the PC will point to upon an event such as a reset (more later)

Vector 0 is the reset vector and tells the CPU what to initialize the PC to after a reset.

We give a label "Entry" that is used in our code as the point to begin executing code

ex) VECTOR 0 Entry

Linker Setup

Assembler Initialization

We use the keyword INIT to tell the Assembler where to begin executing code.

ex) INIT Entry

Passing Parameters/Symbols Between Files

If defined in one file but used in another, we must tell the Assembler these are shared by using the keywords XDEF and XREF

XDEF = "external definition" - used to define a symbol used in another file

XREF = "external reference" - used to signify a symbol used that was defined in another file

If we use XDEF and use the symbol in the same code, we make it visible to the linker
(ex. XDEF Entry)

If we use a Symbol in the Linker Parameter File, we use XREF to use it in our code
(ex. __SEG_END_STACK

Source Code + Linker

Putting It All Together

Source Code (*.asm)

```
            XDEF    Entry
            XREF    __SEG_END_STACK

MyCode:     SECTION
Entry:
            LDS     #__SEG_END_STACK
            :
            :

MyConst:    SECTION
C1          DC.B    1
            :
            :

MyData:     SECTION
D1          DS.B    1
            :
            :
```

Linker Parameter File (*.prm)

```
SECTIONS
            RAM         = READ_WRITE        0X0800 TO 0X0FFF
            ROM_4000    = READ_ONLY         0X4000 TO 0X7FFF
            ROM_C000    = READ_ONLY         0XC000 TO 0XFFFF
END

PLACEMENT
            MyCode, MyConst, DEFAULT_RAM    INTO   ROM_C000
            SSTACK, MyData                  INTO   RAM
END

STACKSIZE 0x0100

VECTOR    0 Entry

INIT      Entry
```

NOTE: - Entry used by linker so we needed XDEF in source code
- By using STACKSIZE, we created the symbol __SEG_END_STACK that could be used in the source code to initialize the stack.

Source Code + Linker

Questions:

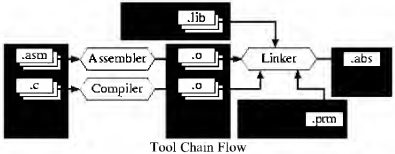
1) Where did the original Linker Parameter File come from in the lab?
How did it know our memory map?

2) What happens if I say:

MyData, SSTACK INTO RAM

Introducing Development Flow and the Parameter File

The Metrowerks CodeWarrior tool-chain is very similar to many others in that it uses a so-called relocatable assembler and compiler, along with a linker. The following figure outlines how the tools work together. The user enters so-called source files that are either assembled or compiled, producing object files. In addition to user supplied object code, most tool-chains provide additional libraries that can be used.



Tool Chain Flow

The linker
follows directives provided by the user in the parameter file to organize the object files and contributions from the library files into a single executable image file. If your only experience with embedded microprocessors involves use of a so called *absolute*
assembler, you will find this process to be more complicated than you are used to. In the long run, however there are benefits to this approach.

Sections and Section Types

Lets contrast the idea of *absolute assembly* with that of *relocatable assembly*. An assembler performs absolute assembly, using the ORG directive which provides the start address of each block of memory. In this way it is known in advance, in the source code, what the actual addresses of code will be. For a short program with few files, written entirely in assembly language, this approach works. However, for larger programs written in a high level language, the use of the ORG directive is less desirable. We don't want to hand tweak so many files.

A relocatable assembler however, defers the assignment of addresses. In combining object files together, the linker is responsible for assigning actual addresses to code and data. With relocating tools you don't specify in your source code the actual addresses. Relocatable tools are useful for writing large programs for large computer systems, where you don't know in advance where code or data will end up. In such systems it is convenient to let the tools deal with such details.

The embedded systems we deal with have characteristics that suggest having absolute control of addressing at some times, and allowing the tools to relocate *things*
at other times. The Metrowerks CodeWarrior tools support both absolute addressing in source code, as well as address relocation. This is done by requiring that regions of code or data, called *sections* be declared as either relocatable or absolute. In the end it is still the linker that positions all the memory sections.

The compiler and assembler organize your code into *sections*. Each section has a *name*, a *type*, and some attributes. The type is either *absolute* or *relocatable*. The section *content*
attribute is assigned by what the section contains. The following outlines the sections types:

Code Section:
A section specifying at least one executable instruction. You will not have *write access* to variables defined in a code section. Additionally, variables in a code section cannot be displayed in the debugger as data.

Constant Section:
A section containing only constant data (variables defined using the DC and DCB directives). It is assumed that such a section corresponds to non-volatile memory and hence there is no need for initialization.

Data Section:
A section containing only non-constant variables. Such data should be allocated into RAM. The stack is saved in a special separate data section.

The Metrowerks assembler and linker
allow a mix of absolute and relocatable sections. The main difference between absolute and relocatable sections involves the way the sections are placed in the system memory space. In using absolute sections, you are responsible for ensuring that no

overlap exists between sections. In using a relocatable section, the linker does the job of assigning addresses according rules outlined in the parameter file.

A Parameter File for Assembler

In the project window click the Link Order tab and double click on `default.prm`. This is the *parameter* file which contains information that the linker
uses to construct executable code. The file is divided into paragraphs and statements. The paragraphs are each opened by the keywords NAMES, SEGMENTS, and PLACEMENT, respectively. Each paragraph is closed with the keyword END. A paragraph is further divided into statements, each of which may end with a semicolon. The other lines in a parameter file are commands. The following is a parameter file written for a small program written entirely in assembly language.

The NAMES paragraph can be used to list object files used to construct the executable code, but generally is not used. Here the file names are passed as command line arguments when the linker is called for execution.

A SEGMENTS
paragraph assigns meaningful names to contiguous memory areas of the target system and can describe a memory map. The segments are named MY_ROM, MY_RAM, and MY_STK. The key words READ_ONLY and READ_WRITE are related to whether or not the segments
are to be initialized. Other key words such as NO_INIT and PAGED are appropriate for battery backed up RAM and paged memory systems, respectively.

default.prm

```
NAMES END

SEGMENTS
  MY_ROM   = READ_ONLY  0x1000 TO 0x1FFF;
  MY_RAM   = READ_WRITE 0x2000 TO 0x2FFF;
  MY_STK   = READ_WRITE 0x3000 TO 0x3FFF;
END

PLACEMENT
  .data     INTO MY_RAM;
  .text     INTO MY_ROM;
  .stack    INTO MY_STK;
END

VECTOR ADDRESS 0xFFFE main

INIT main
```

Contents of `default.prm`

There are cases where the SEGMENTS
paragraph is different from the actual system memory map. In developing an application using RAM, time is saved by not having to reprogram FLASH. In addition, RAM better supports the use of break points.

The PLACEMENT block assigns sections to the named memory blocks. The sections named in a placement paragraph may be user defined, or may be linker-predefined sections. The linker
is case sensitive, each section named here must be a valid predefined or user defined section.

The names `.data`, `.text`, and `.stack` are predefined by the linker
to match sections that are of type data, code, and stack, respectively. The up-shot is that sections named in the source code can be, but are not required to be named in the parameter file placement paragraph. Such unspecified sections will be placed, based on type.

The line beginning with the keyword VECTOR is a command that assigns the address of the instruction at `main` to the reset vector, which is located at address $FFFE. In the source file it is important that the XDEF directive be used to declare `main` as *public*, otherwise the linker
will not be able to find that symbol and thus won't be able to assign the reset vector. Finally, the keyword INIT defines the *entry point* which a simulator uses to start execution of the code.

A Parameter File for 'C'

As a second example parameter file consider that used with the *'Hello World!'* program. Open the project and follow the directions above to open the parameter file. This parameter file is similar to the previous one. The keyword SECTIONS is

synonymous with SEGMENTS, the idea is to describes a kind of memory map. You are responsible for telling the linker what to do.

M68HC912B32_SIM.prm

```
/* This is a generic Prm File - modified for the 68HC12B32 */
NAMES END

SECTIONS
    MY_RAM    = READ_WRITE  0x0800 TO 0x0BFF;
    MY_FLASH  = READ_ONLY   0x8000 TO 0xFFBF;
    MY_EEPROM = READ_WRITE  0x0D00 TO 0x0FFF;
END

PLACEMENT
    DEFAULT_ROM  INTO  MY_FLASH;
    DEFAULT_RAM  INTO  MY_RAM;
END

STACKSIZE  0x100

VECTOR 0 _Startup /* set reset vector on _Startup */
```

Contents of 'C' parameter file

The names DEFAULT_ROM and DEFAULT_RAM in the PLACEMENT paragraph are predefined and understood by the linker to place sections as required in the memory map. For more information than that presented here, refer to the *SmartLinker* manual.

This parameter file shows an alternative way to declare the stack size, here only the stack size is given. The VECTOR command here is similar to that above in that the linker
associates vector number 0 with address 0xFFFE. In this example _Startup refers to a block of code that is executed before `main()` is called.

Written by Jonathan Hill (`jmhill` *at* `hartford.edu`)
Revised: Thu Feb 3 18:49:39 EST 2005

Chapter 5
The Linker

Objectives

This chapter introduces the operation of the linker and gives some specifics about the CodeWarrior® Smart Linker.

5.1 Introduction

A *linker program* combines *object* files to produce an *absolute* file to be loaded into the microcontroller's memory.

In this chapter we investigate the operation of a program used to link together separately assembled and compiled source files into one file. The *linker* produces an output file to be loaded into the microcontroller using the correct types of memory for each portion of the program.

5.2 Assumptions

In this chapter we assume you are using the CodeWarrior® tool set and that you will be running the linker from within the CodeWarrior Integrated Development Environment (IDE). Further, at least until we get to Chapter 8 where we consider C programs, we assume you are developing assembly language programs. In the following sections we describe how the linker locates each of the various parts of an assembly language program into the proper memory.

5.3 Why Use a Relocatable Assembler/Compiler and a Linker?

The relocatable assembler and compiler allow efficient and effective program development using modular programming methodologies much like you use for complex C++ or other high level language programs. You may develop functional modules separately, and independently, from one another and then to *link* them to *build* the final application to be loaded into the microcontroller's memory. Thus a large project can be allocated to different members of the development team. The linker also allows you to have local labels and to expose only a smaller subset of labels to the global program and other modules. Imagine the problems you could have with duplicate labels if you took many modules built by separate groups and tried to assemble them in one giant assembly language file of many thousand lines.

5.4 Memory Types, Sections, and Section Types

In Chapter 2 we described the operation of the relocatable assembler and compiler. We also described the different types of memory, RAM and ROM, and showed that each type is used for different purposes in our embedded system programs. When using the relocatable assembler, the placement of the various parts of our programs in the different memory is called *locating*. That is, we must locate the different parts of the program in the proper memory.

Another task to be done is linking together the separate modules. The need for this arises when code in one module calls or refers to code or data elements in another. This process is also achieved by the linker.

Every program, whether written in assembly language or C, will have various *segments* or *sections*.[1] These are the *code section*, *constant data section, variable data section*, and *stack section*.

Code Section

All parts of the program that must remain when power is turned off, including the *code* and *constant data*, are placed in the code section which is then located in ROM memory.

The code section contains all of the program code. The program MUST be located in read-only memory in an embedded system. In a development environment, where you may have a *development system* or an *evaluation board (EVB)*, you may be place your program and constants into RAM. This makes it easier to change and modify as you are developing and debugging your software. Ultimately, however, you will change the physical location of the code section to ROM for the embedded application.

Constant Data Section

Constants are to be located in ROM too.

Constant data elements are located also in ROM memory. Examples of constant data include variable data initialization values and messages to be printed.

Variable Data Section

The *variable data* section is located in RAM memory.

The variable data section contains the storage locations for all variable data used in the program. There are several points to make about variable data storage used in a program:

- All variable data storage locations must be allocated in the program by the assembler or compiler using a define storage (DS) directive.
- All variable data storage must be located in RAM memory.
- The program code must initialize all variable data must be initialized, if needed, when the processor is running. You MUST not assume any data value will be in a RAM data storage location when the system is power on.

Stack Section

The *stack* section is located also in RAM memory.

As we will see in later chapters, the stack is an allocation of RAM used for temporary information storage. This could be data saved temporarily or return addresses for subroutines. The stack pointer register must be initialized pointing to RAM before the stack can be used.

[1] We will generally use the term section, although you will see segment used in some of the CodeWarrior files. The two terms are synonymous.

5.5 Linker Operation

Locating in Proper Memory

Before we show how to set up the linker to create your program to be loaded into the microcontroller memory, let us have a look at what at what it is supposed to do.

Example 5-1 and Example 5-2 show the *main* and *subroutine* programs for two separately assembled modules. This program does not do anything sensible so we will not worry particularly about the assembly language code.

Main Module

In the main program, Example 5-1, we see there are four distinct uses of memory. The order in which they appear is not important although we will see some programming style suggestions in Chapter 7. The first is a variable data section shown between lines 15 – 17. A memory allocation for a single byte of data storage is made at line 17 by the assembler directive

```
main_data:  DS.B 1.
```

The label `main_data` marks a single byte of storage and will be assigned an address during the linking process. Any other *variable* data storage you might need must be defined in this section or other named sections that can be located in RAM. The section continues until another `SECTION` directive or something other than the define storage (`DS`) directive is found. Everything in this section must be located in RAM memory.

The next memory use is found starting at line 20. This is the *code section* and it starts with the directive

```
MainCode:    SECTION.
```

Everything in this section, from line 20 to the end of the program is to be located in ROM memory. You should choose the label `MainCode` to be something meaningful so you can see where it is located when you view the linker map file.

A *constant data section*, `MainConst: SECTION`, is defined at line 30 and a constant byte is defined by

```
main_const: DC.B $44
```

at line 33. Because constants go into the same type of memory as the code, you could simply define the bytes following the program code. However, it is far better to define a constant section to allow all constants to be grouped together in the linking process or to locate them in a different part of the ROM than the code.

A fourth memory use is implied in line 12 where the label `__SEG_END_SSTACK`[2] is defined to be external. The value of this label will be defined by the linker and is used with the stack pointer initialization instruction at line 22 in the main program. As we will learn later, the stack pointer register must be initialized before calling any subroutines. Also, the stack itself must be located in RAM memory.

---

[2] This label starts with two underbar (_) characters.

Example 5-1 Relocatable "Main" Program

```
Metrowerks HC12-Assembler
(c) COPYRIGHT METROWERKS 1987-2003

 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1
    2    2                      ; Rel_ex_1a_main.asm
    3    3                      ; The main module in a relocatable example.
    4    4                      ; This program does nothing that is sensible.
    5    5                      ; FM Cady 7/04
    6    6                      ; ****************************************
    7    7                      ; Define entry point for the main program.
    8    8                              XDEF  Entry
    9    9                      ; Define symbols used in the subroutine.
   10   10                              XREF  sub_1
   11   11                      ; ****************************************
   12   12                      ; Stack section
   13   13                              XREF  __SEG_END_SSTACK
   14   14                      ; ****************************************
   15   15                      ; Variable Data Section
   16   16                      MainData:   SECTION
   17   17   000000             main_data:  DS.B 1
   18   18                      ; ****************************************
   19   19                      ; code section
   20   20                      MainCode:   SECTION
   21   21                      Entry:
   22   22   000000 CFxx xx              lds   #__SEG_END_SSTACK
   23   23                      ; Initialize the variable main_data.
   24   24   000003 180C xxxx            movb  main_const,main_data
             000007 xxxx
   25   25   000009 B6xx xx     loop:      ldaa  main_data ; Get the data.
   26   26   00000C 16xx xx              jsr   sub_1     ; Pass to sub
   27   27                      ; Increment data for the next loop.
   28   28   00000F 72xx xx              inc   main_data
   29   29   000012 06xx xx              jmp   loop    ; Continue forever.
   30   30                      ; ****************************************
   31   31                      MainConst:  SECTION
   32   32                      ; Define constants in the code section
   33   33   000000 44          main_const: DC.B  $44
```

Subroutine Module

Let us now look at how memory is used in the subroutine shown in Example 5-2. There are three memory uses in this example. There is a section of code starting at line 16 and continuing through line 23. We see a constant data section at line 26 and a variable data section at line 31.

Example 5-2 Relocatable Subroutine

```
Metrowerks HC12-Assembler
(c) COPYRIGHT METROWERKS 1987-2003

 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1                      ; rel_ex_1a_sub_1.asm
    2    2                      ; Relocatable subroutine example with
    3    3                      ; variable and constant data.
    4    4                      ; FM Cady 7/04
    5    5                      ; Input Registers:
    6    6                      ;   A = some input data
    7    7                      ; Output Registers:
    8    8                      ;   A = some output data
    9    9                      ; Registers modified:
   10   10                      ;  A, CCR
   11   11                      ; ****************************************
   12   12                      ; Define symbols used in the subroutine.
   13   13                               XDEF  sub_1
   14   14                      ; ****************************************
   15   15                      ; Code Section
   16   16                      SubCode:  SECTION
   17   17                      sub_1:
   18   18                      ; On entry, the data is in Accumulator A.
   19   19                      ; Store the data in the subroutine.
   20   20   000000 7Axx xx                staa  sub_data
   21   21                      ; Get some data to return to the main.
   22   22   000003 B6xx xx                ldaa  sub_const
   23   23   000006 3D                     rts       ; Return to main program.
   24   24                      ; ****************************************
   25   25                      ; Define constant data.
   26   26                      SubConst: SECTION
   27   27   000000 33          sub_const:DC.B  $33
   28   28                      ; ****************************************
   29   29                      ; Variable Data Section.
   30   30                      SubData:  SECTION
   31   31   000000             sub_data: DS.B  1
```

Memory Needs

Our analysis of memory requirements for the main and subroutine is summarized in Table 5-1. Notice that we have included an estimate of the size of the stack needed for this program. We will defer how we find this out to a later chapter.

Table 5-1 Program Memory Requirements

Module	ROM		RAM	
Main	Code	21 bytes	Data	1 byte
	Constant Data	1 byte	Stack	2 bytes
Subroutine	Code	7 bytes	Data	1 byte
	Constant Data	1 byte		

The Hardware Memory Map

As programmers, now we know how much ROM and RAM our program will require. As embedded system engineers, we must look at the hardware upon which our program is going to be installed. We do that by looking at the *memory map* of the system. In a single-chip microcontroller, such as we described in Chapter 2, the manufacturer determines the memory map when the chip is designed and manufactured. Figure 5-1 shows a MC9S12C32 microcontroller operating in *single*-chip mode in an embedded system with no external RAM memory.

Software engineers must know the system memory map to be able to load their code into the microcontroller's memory.

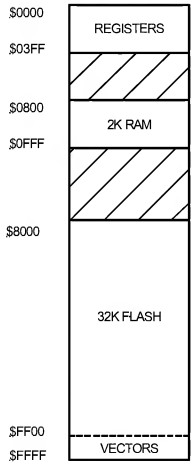


Figure 5-1 Memory Map for MC9S12C32 Microcontroller

Linking the Code

Before we show how to instruct the linker to locate our code correctly, we need to discuss the linking part of this job. Linking refers to establishing the final address for certain instructions, such as a *jsr – jump-to-subroutine*, when the starting address of the subroutine is in another module. This address resolution must be performed also when one module refers to a data location defined in another module. Line 26 in Example 5-1 has a `jsr sub_1` instruction where `sub_1` is in the second, separately assembled module. The assembler has three directives that organize the information needed to accomplish this linking task.

XREF, XDEF, and XREFB

When a symbol is used in one module but is defined in another, it is said to be an *external* symbol. In this case, the *XREF – External Reference* directive is used. In the module where the symbol is defined, the *XDEF – External Definition* directive is used. In Example 5-1 the symbol `sub_1` is the address for the jump-to-subroutine instruction on line 26. Because that symbol is not defined in the main module, it is declared to be external by `XREF sub_1` on line 10. Similarly, the `sub_1` symbol must be made visible in the subroutine module. This is done by the `XDEF sub_1` on line 13. For every XREF in our program modules there must be a partner XDEF in some other module.[3]

Every XREF must have a companion XDEF in some other module.

Example 5-1 shows `XDEF Entry` at line 8. This makes the code entry point (where the `Entry` label is at line 21) visible to the linker.

There is another directive, *XREFB – External Reference for Symbols on the Direct Page* included in the assembler directives. This is used when an external symbol is on the direct addressing page and thus fewer addressing bits are required.

5.6 The Linker Parameter File

The linker is controlled by a *linker parameter file (*.prm)*. You put the memory map information shown in Figure 5-1 in the linker parameter file so the linker can locate each section properly. Example 5-3 shows a linker parameter file used to link (and locate) the two modules in Example 5-1 and Example 5-2.

The linker parameter file allows us to specify easily where code and data are located.

Example 5-3 Linker Parameter File

```
NAMES
END

SECTIONS
    RAM = READ_WRITE 0x0800 TO 0x0FFF;
/* unbanked FLASH ROM */
    ROM_C000 = READ_ONLY  0xC000 TO 0xFEFF;
END

PLACEMENT
/* Place the code and constant sections in ROM */
    .init, MainCode, SubCode,MainConst, SubConst,
    DEFAULT_ROM  INTO  ROM_C000;
/* Place the Variable Data */
    MyData, MySubData,DEFAULT_RAM  INTO  RAM;
END
/* Define the stack size needed */
STACKSIZE 0x100
/* Specify the initial reset vector entry point */
VECTOR 0 Entry  /* reset vector */
/* INIT is needed for assembly applications */
INIT Entry
```

[3] If you forget to declare a symbol as external with XREF, the assembler will give you an error message. If you forget the XDEF, the linker will generate an error message.

Parameter File Commands

Although the CodeWarrior linker is very powerful with many features, we will describe only those few necessary to link assembly and C program modules. Keywords and commands in the parameter file must be capitalized. Each section in the parameter file starts with the command name in capitals and ends with the END. All commands and qualifiers are case sensitive.

Names:

The *NAMES* command lists the files building the application. For an application linked in the CodeWarrior tool set, the files that have been "added" to the application are automatically linked. Other modules may be linked by including their names in the NAMES section.

Sections:

```
SECTIONS
    RAM = READ_WRITE 0x0800 TO 0x0FFF;
/* unbanked FLASH ROM */
    ROM_C000 = READ_ONLY  0xC000 TO 0xFEFF;
END
```

The *SECTIONS* part defines the memory map for your application. It allows you to assign meaningful names to address ranges. These can be used in the subsequent placement section to increase the readability of your parameter file. The syntax of a SECTIONS entry is the following:

```
<your_name> = MEMORY_TYPE  address TO address;
```

where

`<your_name>` is the name you wish to use for the particular address range, and `MEMORY_TYPE` defines the type of memory as shown by the qualifiers in Table 5-2.

Table 5-2 Memory Type Qualifiers

Qualifier	Initialized Variables	Non-Initialized Variables	Constants	Code
READ_ONLY	Not applicable.[4]	Not applicable.[4]	Content written to target address.	Content written to target address.
READ_WRITE	Content written into copy down area with information to allow it to be copied at startup.[5]	Area contained in zero out information.[6]	Content written into copy down area.[5,6]	To allocate code in a RAM area for system development, you should declare this area as READ_ONLY. In an embedded system, this is not applicable.[4]
NO_INIT	Not applicable.[4]	Handled as allocated.	Not applicable.[4]	Not applicable.[4]
PAGED	Not applicable.[4]	Handled as allocated.	Not applicable.[4]	Not applicable.[4]

[4] These cases are not intended but the linker does allow them. When used, the qualifier controls what is written into the application.

[5] Initialized objects and constants in READ_WRITE must be initialized at the start of the program at run-time. In a C program this is done by the start-up code but in an assembly language program you must ensure this is done. The copy down area contains information to initialize the constants in the start-up code.

[6] The zero out information specifies areas that must be initialized with 0 at startup.

Placement:

```
PLACEMENT
/* Place the code and constant sections in ROM */
    .init, MainCode, SubCode, MainConst, SubConst,
    DEFAULT_ROM  INTO  ROM_C000;
/* Place the Variable Data */
    MyData, MySubData, DEFAULT_RAM  INTO  RAM;
END
```

The *PLACEMENT* section places your various sections into the proper type of memory as defined in the SECTIONS section. You must include `DEFAULT_RAM` and `DEFAULT_ROM` placements. The linker, by default, will place your sections into the proper placement. For example, the linker knows to place any memory allocation (DS) parts of the program into DEFAULT_RAM and any code and constants (DC) into DEFAULT_ROM. You may, for the sake of documentation readability or to control where sections are placed, "place" the named sections from your program as shown in Example 5-3. This is optional but is a good documentation practice.

Stacksize:

```
/* Define the stack size needed */
STACKSIZE 0x100
```

An important part of your code development is the allocation of storage and placement of the stack. As we will learn in Chapter 6, the stack is an area of RAM used for temporary variable storage and for return addresses for subroutines. Because both the stack and the variable data storage sections are in READ_WRITE memory, it is important that neither one interfere with or overwrite the other. You must estimate the amount of stack storage that is needed and specify at least this amount. Because stack overflow (using more stack memory than is allocated) can cause many problems in your running program, most embedded system engineers are very conservative and allocate plenty of stack space. In this example, `STACKSIZE` allocates $0100 bytes, places the stack into RAM and defines the variable `__SEG_END_SSTACK` to be used in your program to initialize the stack pointer register (as shown in line 22 in Example 5-1.)

Vector:

```
/* Specify the initial reset vector entry point */
VECTOR 0 Entry  /* reset vector */
```

The vector section allows you to initialize vectors. This example shows how to initialize the reset vector to allow the microcomputer to be *vectored* to the start of the program when power is turned on. We will see other vectors and their uses for interrupts in Chapter 12.

Init:

```
/* INIT is needed for assembly applications */
INIT Entry
```

When linking an assembly application, an *entry point* is required. The *INIT* command with its operand, in this case *Entry*, provides this information.

Comments:

Comments are entered into the parameter file using standard C or C++ syntax for comments.

5.7 The Linker Output Files

Two output files created by the linker are important to us after we have located and linked our object files. These are the linker map file, which shows us where the linker has placed our code, and the absolute code file to be loaded into the microcontroller's memory.

The Linker Map File

The *Linker Map File* shows us where our code, constant and variable data are located, along with other information. Following the linking controlled by the linker parameter file in Example 5-3, the linker map file shows the following sections.

The *linker map file* shows us where all elements of our program are located.

Target Section:

```
TARGET SECTION
-------------------------------------------------------------------------------------------------
Processor    : Motorola HC12
Memory Model: SMALL
File Format : ELF\Dwarf 2.0
Linker       : SmartLinker V-5.0.22 Build 4047, Feb 17 2004
```

The target section shows the processor, the memory model (more important when programming in C), the file format, and the linker version.

File Section:

```
FILE SECTION
-------------------------------------------------------------------------------------------------
rel_ex_1a_main.asm.o                  Model: SMALL,          Lang: Assembler
rel_ex_1a_sub_1.asm.o                 Model: SMALL,          Lang: Assembler
```

This lists all files that have been linked.

Startup Section:

```
STARTUP SECTION
-------------------------------------------------------------------------------------------------
Entry point: 0xC000 (Entry)
```

The startup section shows the entry point, i.e. where the program will start.

Section-Allocation Section:

```
SECTION-ALLOCATION SECTION
Section Name                          Size  Type     From      To        Segment
-------------------------------------------------------------------------------------------------
.init                                   21     R     0xC000    0xC014    ROM_C000
SubCode                                  7     R     0xC015    0xC01B    ROM_C000
MainConst                                1     R     0xC01C    0xC01C    ROM_C000
SubConst                                 1     R     0xC01D    0xC01D    ROM_C000
MainData                                 1   R/W      0x800     0x800    RAM
SubData                                  1   R/W      0x801     0x801    RAM
.stack                                 256   R/W      0x802     0x901    RAM
.vectSeg0_vect                           2     R     0xFFFE    0xFFFF    .vectSeg0

Summary of section sizes per section type:
READ_ONLY (R):          20 (dec:       32)
READ_WRITE (R/W):      102 (dec:      258)
```

Each of the software components that have been allocated into the various sections is described. Note that the code in the module that is first executed, that is the main module, is placed into a section named *.init*. In a C program, the startup code will be placed in .init. All other named sections, such as SubCode (in the subroutine), MainConst, MainData and Subdata are listed with their size, type of memory, the range of address, and the segment assignment.

Vector-Allocation Section:

```
VECTOR-ALLOCATION SECTION
    Address      InitValue   InitFunction
-------------------------------------------------------------------------------

    0xFFFE         0xC000    Entry
```

This section shows how vectors have been initialized. We will see more of this when we discuss interrupts in Chapter 12.

Object-Allocation Section:

```
OBJECT-ALLOCATION SECTION
    Name                  Module                  Addr   hSize   dSize     Ref Section  RLIB
-------------------------------------------------------------------------------------------
MODULE:                -- rel_ex_1a_main.asm.o --
- PROCEDURES:
    Entry                                         C000       9       9       0 .init
    loop                                          C009       C      12       1 .init
- VARIABLES:
    main_data                                      800       1       1       3 MainData
    main_const                                    C01C       1       1       1 MainConst
- LABELS:
    _SEG_END_SSTACK                                902       0       0       1
MODULE:                -- rel_ex_1a_sub_1.asm.o --
- PROCEDURES:
    sub_1                                         C015       7       7       1 SubCode
- VARIABLES:
    sub_const                                     C01D       1       1       1 SubConst
    sub_data                                       801       1       1       1 SubData
```

One of the most important sections is the object-allocation section. It shows, for each of the modules (in this case the main and subroutine) where the procedures, labels, and data elements are located. We will need this information when it comes time to debug our program. The hSize and dSize give the number of bytes in hexadecimal and decimal.

Module Statistic:

```
MODULE STATISTIC
  Name                                      Data    Code  Const
-------------------------------------------------------------------------------

  rel_ex_1a_main.asm.o                         2      42      2
  rel_ex_1a_sub_1.asm.o                        2      14      2
  other                                      256       2      0
```

This shows for each module the number of code, data, and constants bytes that have been allocated. In this listing, *other* refers to the space allocated to the stack.

Section Use in Object-Allocation Section:

```
SECTION USE IN OBJECT-ALLOCATION SECTION
SECTION: ".init"
  Entry loop
SECTION: "SubCode"
  sub_1
SECTION: "MainConst"
  main_const
SECTION: "SubConst"
  sub_const
SECTION: "MainData"
  main_data
SECTION: "SubData"
  sub_data
```

This section shows which elements (or objects) of our program use which memory section.

Object List Sorted by Address:

```
OBJECT LIST SORTED BY ADDRESS
     Name                                      Addr    hSize    dSize     Ref Section RLIB
-------------------------------------------------------------------------------------------
     main_data                                  800        1        1       3 MainData
     sub_data                                   801        1        1       1 SubData
     Entry                                     C000        9        9       0 .init
     loop                                      C009        C       12       1 .init
     sub_1                                     C015        7        7       1 SubCode
     main_const                                C01C        1        1       1 MainConst
     sub_const                                 C01D        1        1       1 SubConst
```

The address of each object (denoted by a label) is given here.

Unused Object Section:

This lists any objects found in the object files that were not linked, such as a label not referenced.

Copydown Section:

This section lists all blocks that are copied from ROM to RAM at program startup. This is more relevant in C programming.

Object-Dependencies Section:

```
OBJECT-DEPENDENCIES SECTION
-------------------------------------------------------------------------------
Entry                     USES __SEG_END_SSTACK main_const main_data
loop                      USES main_data sub_1 loop
sub_1                     USES sub_data sub_const
```

This section lists the names of global objects which every function and variable.

Dependency Tree:

An object dependency tree shows in a tree format all detected dependencies between functions.

Statistic Section:

```
STATISTIC SECTION
-------------------------------------------------------------------------------
ExeFile:
--------
Number of blocks to be downloaded: 5
Total size of all blocks to be downloaded: 32
```

This section gives the number of bytes of code in the application.

Absolute Files

The linker can produce two kinds of output files to be loaded in the microcontroller memory. These are absolute, *.abs*, and Motorola S files.

Absolute files, which have the extension .abs, contain the program code plus some debugging information. We will see how to use this information in Chapter 8. Motorola S files, which have extensions .s1, .s2, .s3 or .s19, contain all code from the READ_ONLY sections of the application. This, then, can be downloaded to the microcontroller or programmed into the ROM.

5.8 Remaining Questions

In this chapter we have described enough of the features of the linker to get started in assembly language programming. We still have a number of issues to cover and we will

do that in succeeding chapters. Some of these details yet to be explained include the following topics:

- What is startup code? *Startup code is the code that is executed when the processor is first powered-up or the program is first run. In C programs a standard startup code sequence is used. We will discuss this issue further in Chapter 8.*

5.9 Conclusion and Chapter Summary Points

In this chapter we have discussed the operation of the linker. We have included enough details to be able to assemble and link assembly language programs using separately assembled relocatable modules.

- There are two kinds of memory in a microcontroller system – RAM and ROM.
- There can be two types of ROM – EEPROM and FLASH EEPROM.
- You must know what the memory map is to be able to locate the various parts of the program in the proper memory types.
- Variable data storage and the stack use RAM.
- The program code and constants use ROM.
- The linker is controlled by a linker parameter file.
- It is very easy to change the parameter file to relocate the code for different hardware configurations.

5.10 Bibliography and Further Reading

Cady, F. M., *Microcontrollers and Microprocessors*, Oxford University Press, New York, 1997.

Smart Linker, Metrowerks Corporation, Austin, TX, 2003.

5.11 Problems

Table of Contents

Chapter Listings:
Revision Chapters

1. Introduction
2. General Principles of Microcontrollers
3. Introduction to the M68HC(S)12 Hardware
4. An Assembler Program – CodeWarrior
5. A Linker Program
6. The M68HC12 Instruction Set
7. Assembly Language Programs for the M68HCS12
8. Simulating M68HCS12 Programs
9. Debugging M68HCS12 Programs
10. Program Development Using C
11. M68HCS12 Parallel I/O
12. M68HCS12 Interrupts
13. M68HCS12 Memories
14. M68HCS12 Timer
15. M68HCS12 Serial I/O – SCI, SPI
16. M68HCS12 Serial I/O – CAN, IIC
17. M68HCS12 Analog Input
18. Single-Chip Microcomputer Interfacing Techniques
19. Fuzzy Logic
20. Debugging Systems
21. Advanced M68HCS12 Hardware
22. Appendix A – D-Bug12 Monitor
23. Appendix B – Debugging Systems Pod Design
24. Appendix C – Notations and References